Jose Revete
19-10040

Tarea 3

1. **Lenguaje escogido: JAVA**
    a. **De una breve descripción del lenguaje escogido.**
        i. **Diga qué tipos de datos posee y qué mecanismos ofrece para la creación de nuevos tipos (incluyendo tipos polimórficos de haberlos).**

R:

Tipos simples

- **Predefinidos:**
    - Booleanos: (true o false).
    - Caracteres.
    - Numéricos:
        - Enteros: byte, short, int, long.
        - Flotantes: float, double.
- **Enumeraciones**:
  Usando la palabra clave enum, se pueden definir tipos con un conjunto fijo de valores constantes. Las enumeraciones también pueden contener métodos, atributos y constructores.

```
enum Dias { LUNES, MARTES, MIERCOLES; }
```

- **Sub-rangos**:
  Se pueden **simular** con clases que validen el rango permitido:

```
class SubRango {
    private int valor;

    public SubRango(int valor) {
        if (valor < 1 || valor > 100) throw new
IllegalArgumentException("Fuera de rango");
        this.valor = valor;
    }

    public int getValor() { return valor; }
}
```

- **Tipo vacío (void) y unitario**:
    - **void** se usa como tipo de retorno en métodos que no devuelven datos.
    - El tipo **unitario** no existe directamente en Java, pero es posible **simularlo** usando enum con un solo valor.

Tipos compuestos

- **Registros**:
  Permiten definir tipos inmutables que encapsulan datos con menos código.

```
record Punto(int x, int y) {}
```

- **Arreglos y listas**:
  - **Arreglos**: Ejemplo: `int[] numeros = {1, 2, 3};`
  - **Listas**: Usan clases del *framework* de colecciones como *ArrayList* o *LinkedList*.
- **Conjuntos**:
  Es posible **implementarlo** mediante clases como *HashSet* o *TreeSet*.
- **Apuntadores y tipos recursivos**:
  Java no tiene apuntadores explícitos, **pero las referencias a objetos permiten construir estructuras recursivas** como listas enlazadas o árboles:

```
class Nodo {
    int valor;
    Nodo siguiente;
}
```

Mecanismos para crear nuevos tipos

- **Clases**:
  Las clases son la base para definir nuevos tipos. Pueden contener atributos, métodos y constructores:

```
class Persona {
    String nombre;
    int edad;

    Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }}
```

- **Interfaces**:
  Permiten definir contratos que otras clases deben implementar. Son esenciales para el polimorfismo:

```
interface Animal {
    void hacerSonido();
}
```

```
class Perro implements Animal {
    public void hacerSonido() {
        System.out.println("Guau");
    }
}
```

- **Herencia y polimorfismo**:
  Java permite la herencia de clases (*extends*) para crear jerarquías reutilizables. El polimorfismo se logra mediante el uso de referencias de la *superclase* o *interfaces*.
- **Generics**:
  Usados para crear clases y métodos parametrizados con tipos, como *List<T>*:

```
class Caja<T> {
    private T contenido;
    public void guardar(T item) { contenido = item; }
    public T obtener() { return contenido; }
}
```

- **Clases internas y anónimas**:
  - Internas: Definidas dentro de otra clase.
  - Anónimas: Declaradas e instanciadas en una sola expresión para implementar clases o interfaces rápidamente.

---

ii. Describa el funcionamiento del sistema de tipos del lenguaje, incluyendo el tipo de equivalencia para sus tipos, reglas de compatibilidad y capacidades de inferencia de tipos.

R:

Sistema de tipos en Java

Java verifica los tipos en tiempo de compilación y requiere definiciones explícitas en la mayoría de los casos (dado que es tipado estático, fuerte y basado en nombres).

- **Equivalencia de tipos**
  - Equivalencia por nombres: Java **utiliza principalmente la equivalencia por nombres** para determinar si dos tipos son iguales. Ejemplo: Aunque dos clases tengan los mismos atributos, son incompatibles si sus nombres son distintos.
  - Alias de tipos: Java **no permite** alias explícitos de tipos.
  - Subtipos: Java admite equivalencia parcial basada en la jerarquía de clases.

```
class Animal {}
```

```
class Perro extends Animal {}
Animal a = new Perro();
```

- **Reglas de compatibilidad**
  - Tipos primitivos:
    - Soporta promoción automática (tipos más pequeños pueden asignarse a tipos más grandes sin conversión explícita). Por ejemplo: int a long.
    - Requiere casting explícito al convertir a un tipo más pequeño o diferente.
  - Tipos referenciados:
    - Herencia permite la compatibilidad entre superclases y subclases, aunque el casting puede ser necesario.
    - Los genéricos utilizan eliminación de tipos, por lo que no son compatibles en tiempo de ejecución sin un casting explícito.

```
List<String> lista = new ArrayList<>();
List rawList = lista;
```

- **Inferencia de tipos**
  - Genéricos: Deduce el tipo de colecciones y métodos a partir del contexto.

```
List<String> lista = List.of("a", "b");
```

  - var: Permite inferir el tipo de variables locales:

```
var mensaje = "Hola";
```

  - Métodos genéricos: Java infiere los tipos genéricos según los argumentos:

```
<T> void imprimir(T elemento) {
System.out.println(elemento); }
imprimir(42);
```

---

iii. Diga si la totalidad de los tipos de datos del lenguaje conforman un álgebra. Diga si posee construcciones que corresponden a: el tipo producto, el tipo suma, el tipo cero (neutro de la suma) y el tipo uno (neutro del producto).

R:

Sí, los tipos de datos en Java **pueden** analizarse como un sistema algebraico. **Aunque no está diseñado formalmente como tal**, Java tiene construcciones que corresponden a los conceptos de:

- **Tipo producto**: Representado por clases y registros, que agrupan múltiples valores en una sola entidad.
- **Tipo suma**: Simulado con herencia, polimorfismo e enum, permitiendo representar alternativas.
- **Tipo cero (neutro de la suma)**: Corresponde a null, que indica la ausencia de valor en tipos no primitivos.

- **Tipo uno (neutro del producto)**: Representado por void, que tiene un único valor implícito: no retorna nada.

4. Realice un análisis de los tiempos obtenidos y de un razonamiento sobre el comportamiento observado.

R:

Con respecto al comportamiento observado, podemos ver que mientras más grande sea M y N, más tardará el programa en recorrer la matriz. También que es necesario liberar la mejora ocupada por matriz luego de cada iteración, dado que sin ello, obtendremos errores.

Además, en las implementaciones de recorre primero por fila y luego por columna, y viceversa, los tiempos de ejecución son muy similares, sin embargo, en rasgos generales observamos que el recorrido primero por fila y luego por columna en minimamente más eficiente que su opuesto, pero dicha diferencia aumenta cada vez más que el tamaño de M y N aumenta, por ejemplo: en 100x1000 la diferencia es de 0.00001 seg, pero en 1000000x1000000 la diferencia es de 2.837646 seg. A su vez, podemos intuir que el tamaño de la matriz afecta directamente al tiempo de ejecución, mas no la forma de este (si es cuadrada, si N>M o N<M).

Otro aspecto importante es que los tiempos de ejecución siempre son distintos al ejecutar la misma configuración de matriz, las variaciones no son lo suficientemente resaltables, pero no dejan de ser notorios.

En cuanto a la interrogante de si los tiempos de ejecución se ven afectados si la matriz de declara de forma global o local, obtendremos que la forma en que se declara y se asigna memoria para la matriz puede afectar los tiempos de ejecución debido a las diferencias en la gestión de memoria y el acceso a la memoria. Sin embargo a pesar de que las pruebas realizadas con global son con N y M realmente bajos para realizar las comparaciones, encontramos que local en promedio es más rápido en tiempo de ejecución que global, aun así, queda una gran franja gris que es el comportamiento de global con nuevos más grandes para N y M (no se realizó dado que ocurre un desbordamiento de memoria).